4-1998

Class-Based and Algebraic Models of Objects

Gary T. Leavens
Iowa State University

Don Pigozzi
Iowa State University

Class-Based and Algebraic Models of Objects

Abstract
We compare different kinds of first-order models of objects and message passing, as found in object-oriented programming languages. We show that generic function models can easily simulate record models for static, class-based languages. We explore type systems for such languages, and show that our simulation preserves typing. Algebraic models emerge as abstractions of the generic function model that suppress details that are irrelevant for client code.

Keywords
object-oriented, record, generic function, type, subtype, algebraic model, category-sorted algebra, order-sorted algebra, semantics, type theory

Disciplines
Systems Architecture | Theory and Algorithms

Class-Based and Algebraic Models of Objects

Gary T. Leavens and Don Pigozzi
TR #98-02
April 1998

Keywords: object-oriented, record, generic function, type, subtype, algebraic model, category-sorted algebra, order-sorted algebra, semantics, type theory.

1998 CR Categories: D.3.1 [*Programming Languages*] Formal Definitions and Theory — Semantics; D.3.3 [*Programming Languages*] Language Constructs — Abstract data types; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — algebraic approaches to semantics, denotational sematics.

Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Class-Based and Algebraic Models of Objects

Gary T. Leavens*
Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, Iowa 50011 USA
Email: leavens@cs.iastate.edu

Don Pigozzi
Iowa State University
Department of Mathematics, Iowa State University
416 Carver Hall, Ames, Iowa 50011 USA
Email: dpigozzi@iastate.edu

April 6, 1998

Abstract

We compare different kinds of first-order models of objects and message passing, as found in object-oriented programming languages. We show that generic function models can easily simulate record models for static, class-based languages. We explore type systems for such languages, and show that our simulation preserves typing. Algebraic models emerge as abstractions of the generic function model that suppress details that are irrelevant for client code.

1 Introduction

Along with the promise of reuse, object-oriented (OO) techniques bring several challenges. A key problem that our research addresses is how to verify (or reason about) code that uses message passing and subtype polymorphism.

Our research on such questions has been mostly model-theoretic [18, 29, 30, 31, 32, 33, 34, 35, 36]. However, the models we use may seem, at first glance, to have little to do with standard OO programming languages, such as Smalltalk-80 [24], C++ [50], Eiffel [42] and Java [2, 25]. Such single-dispatching languages seem to be better modeled by models in which objects resemble records, and message passing is modeled by looking up a method in the object record. By contrast, the models we use resemble multiple-dispatching OO languages such as CLOS [44], Dylan [48], and Cecil [11, 12].

*The work of both authors was supported in part by NSF grant CCR-9593168.

In this paper we describe and relate these various kinds of models. In doing so we also establish some connections between class-based single-dispatching and multiple-dispatching OO languages. We also describe the ways in which our models are abstractions of the semantics of multiple-dispatching OO programming languages. In this way we hope to make clear the connection between class-based single-dispatching OO languages and our work.

Our research has concentrated on the verification of client code. *Client code* manipulates objects by sending them messages. Unlike the code used to implement OO classes, client code does not access the internal fields of objects. Client code is thus insulated from changes in to the internal details of objects.

In this paper we aim to help readers relate our models to the semantics of the most familiar OO languages. To that end we explore the semantics of client code in such class-based single-dispatching languages and relate them to the semantics of multiple-dispatching languages in the first section below. In the following section we relate type systems appropriate for the two kinds of models. Following that, we relate the semantics of multiple-dispatching languages to our algebraic models. For those more familiar with such models, we also relate our style of model to order-sorted and category-sorted models.

2 Semantics of Objects and Message Passing

Many semantics of OO languages have appeared in the literature. (See Abadi and Cardelli's book [1] and Castagna's book [9] for surveys.) Since we are concerned with client code, we can largely avoid the knotty semantical problems of modeling inheritance (see, for example [3, 15, 16, 28]). Instead, we focus on the semantics of objects and message passing from the client's point-of-view. Our aim is to relate the semantics of client expressions in class-based single-dispatching languages to those in multiple-dispatching languages. We start with the class-based single-dispatching languages.

2.1 Record-based Models

One way to model objects in an OO language is as a record containing data and method (operation) fields [1, 7]. Although this model of objects is somewhat naive [1, Section 6.7], it has the virtue of familiarity and simplicity. This kind of model is most appropriate for prototype languages such as Self [49, 51] and others [19, 37]. The inclusion of methods in objects allows great flexibility, since, for example, a program can create an unbounded number of objects, each with different methods.

Including methods in objects is somewhat of an abstraction of the most popular class-based languages, Smalltalk-80, C++, Eiffel, and Java. In such class-based languages methods are typically accessed indirectly through a class pointer, because all objects of the same class share the same methods. Furthermore, we will ignore Smalltalk and Java's ability to define new classes at run-time, so that the set of classes (and hence methods) is statically deter-

minable. We calls such languages *static, class-based* languages. We study static, class-based languages in this paper because they can be simulated by multiple-dispatch generic function languages, which are closer to our algebraic models. (Furthermore, prototype-based languages have been very extensively studied already [1].) The reader should keep in mind, however, that we are limiting ourselves to a subset of the single-dispatching languages.

We now describe models for static, class-based single-dispatching languages in detail.

Ignoring the possibility of mutation, a record model tailored to static, class-based languages can be defined with the following semantic domains. (As usual, we list our abbreviations for typical elements to the left.) Objects are records, and records are themselves modeled as finite functions from a domain of record labels to either data or methods.

$$
\begin{array}{lll}
(\rho_r, \rho_c) \in & \text{REnvironment} & = (\text{Identifier} \xrightarrow{\text{fin}} \text{Data}) \times (\text{ClassId} \xrightarrow{\text{fin}} \text{RMethDict}) \\
I \in & \text{Identifier} & \\
d \in & \text{Data} & = \text{Int} + \text{Bool} + \text{Object} + \text{Data}^* \\
o \in & \text{Object} & = \text{ClassId} \times \text{Record} \\
r \in & \text{Record} & = (\text{Label} \xrightarrow{\text{fin}} \text{Data}) \\
l, g \in & \text{Label} & = \text{Identifier} \\
t \in & \text{ClassId} & = \text{Identifier} \\
c \in & \text{RMethDict} & = (\text{Label} \xrightarrow{\text{fin}} \text{Method}) \\
m \in & \text{Method} & = \text{Data} \to \text{Data}_\perp
\end{array}
$$

A ClassId is just a name; the second (ρ_c) part of the environment is used to map each such class name to its method dictionary. Method dictionaries map method names (labels) to methods [52]. The class of an object is contained in the object itself. Hence sending a message in this model means selecting a method from an object's class, using the method's label, and calling the code that is found as in a procedure call. (Note also that no concurrency is necessarily involved.)

The syntax of a message send expression is $E_0\texttt{.}l\texttt{(}E_1\texttt{)}$. Semantically, this applies the function $E_0\texttt{.}l$ to the argument E_1. In the jargon, however, this is thought of as sending the message $l\texttt{(}E_1\texttt{)}$, with message name l and argument $\texttt{(}E_1\texttt{)}$ to the object denoted by E_0.

Besides the explicit argument passed in a message, a method also has access to the object that is being sent the message (E_0 in $E_0\texttt{.}l\texttt{(}E_1\texttt{)}$). This object is called `self` in Smalltalk, and `this` in Java. It is also called the *implicit* or *default argument* of a method. A method obtains access to `self` in one of two ways [1, Section 6.7].

- Methods can be constructed as fixpoints of premethods (functionals that take `self` as an argument) [16, 15, 28], which builds in `self`.
- Methods can be explicitly passed `self` as the method's first argument when called [43].

These two variations turn out to be equivalent [5], although the first variation has problems in explaining method update [1, Section 6.7.2]. For our purposes the second variation is more convenient. We therefore give the following semantics for identifier, message send, and tupling expressions. In this semantics, the environment is written as (ρ_r, ρ_c) and we ignore mutation. (The typographical conventions used is this semantics are from Schmidt's book [47]. The **cases** expression is used for disjoint union types, with functions of the form "in$X$" being injections into the disjoint union from the domain X and "is$X$" being a test to see if an element of the disjoint union was injected from X. Pattern matching with is$X$ is also used, and binds data to the names given, as in ML and Haskell. For example, consider the following formula.

cases inInt(1) **of**
    isInt(j) $\rightarrow$ $j + j$
    else $\rightarrow$ 3
end

This has the value 2.)

$\mathcal{E}_r$: Expression $\rightarrow$ REnvironment $\rightarrow$ Data$_\perp$
$\mathcal{E}_r[\![I]\!](\rho_r, \rho_c) = \rho_r(I)$
$\mathcal{E}_r[\![E_0\,.\,l\,(E_1)]\!](\rho_r, \rho_c) =$
    cases $\mathcal{E}_r[\![E_0]\!](\rho_r, \rho_c)$ **of**
        isObject(I, r) $\rightarrow$ $\rho_c(I)(l)(\text{inObject}(I, r), \mathcal{E}_r[\![E_1]\!](\rho_r, \rho_c))$
        else $\rightarrow$ $\perp$
    end
$\mathcal{E}_r[\![(E_1, \ldots, E_n)]\!](\rho_r, \rho_c) = (\mathcal{E}_r[\![E_1]\!](\rho_r, \rho_c), \ldots, \mathcal{E}_r[\![E_n]\!](\rho_r, \rho_c))$

These three kinds of expressions we call *client expressions*. Other client expressions could easily be added, but we specifically prohibit a client expression from directly extracting the data fields of an object; this prohibition promotes information hiding.

2.2 Generic Function Models

A second way to model objects is an abstraction of multiple-dispatching OO languages such as CLOS, Dylan, and Cecil. In this kind of model, objects only contain data, not methods. The methods are moved outside the object [9, 10]. All methods with the same name are grouped into a *generic function*, which also has that name. Hence we call this kind of model a generic function model.

To be symmetric with the class-based record model presented above, we present a class-based generic function model as well. This is again a restriction in the space of generic function languages; for example, Cecil is not class-based. Our model also ignores object identity and the possibility of mutation. The Data, Object, Record, Method, Label, ClassId, and Identifier domains are ex-

actly the same as in the record model, but are repeated here for convenience.

$$
\begin{array}{lll}
(\rho_d, \rho_f) \in & \text{GEnvironment} & = (\text{Identifier} \xrightarrow{\text{fin}} \text{Data}) \times (\text{Label} \xrightarrow{\text{fin}} \text{GGenFun}) \\
I \in & \text{Identifier} & \\
d \in & \text{Data} & = \text{Int} + \text{Bool} + \text{Object} + \text{Data}^* \\
o \in & \text{Object} & = \text{ClassId} \times \text{Record} \\
r \in & \text{Record} & = \text{Label} \xrightarrow{\text{fin}} \text{Data} \\
l, g \in & \text{Label} & = \text{Identifier} \\
I \in & \text{ClassId} & = \text{Identifier} \\
ct \in & \text{ClassIdTree} & = \text{ClassId} + \text{ClassIdTree}^* \\
f \in & \text{GGenFun} & = \text{ClassIdTree} \xrightarrow{\text{fin}} \text{Method} \\
m \in & \text{Method} & = \text{Data} \to \text{Data}_\perp
\end{array}
$$

As in the record model, environments are composed of two parts. The main difference is the ways the second part of the environment is organized. In the generic function model, the second part (ρ_f) groups all methods with the same name into a generic function, which can be used to select a method based on the class of the argument. This is inverted from the record model, where the second part of the environment groups methods by class, and uses the method name to select the method from a class's method dictionary.

In the generic function model, message passing means selecting a method from a generic function and calling it. Method selection from a generic function is based on the classes of arguments of a message, which may, in general, be trees of class names. Formally our model of such trees is given by the domain ClassIdTree above. We reserve the class names `int` and `bool` for the built-in types. This allows us to define the class tree for a data element with the following (strict) function.

$$
\begin{array}{l}
\mathit{classOf}\colon \text{Data}_\perp \to \text{ClassIdTree}_\perp \\
\mathit{classOf}(\text{inInt}(i)) = \text{inClassId}(\texttt{int}) \\
\mathit{classOf}(\text{inBool}(b)) = \text{inClassId}(\texttt{bool}) \\
\mathit{classOf}(\text{inObject}(I, r)) = \text{inClassId}(I) \\
\mathit{classOf}(\text{inData}(d_1, \ldots, d_n)) = \text{inClassIdTree}^*(\mathit{classOf}(d_1), \ldots, \mathit{classOf}(d_n))
\end{array}
$$

We use tuple notation to abbreviate these trees. That is, we write I for $\text{inClassId}(I)$, $(I_1, \ldots, I_n)$ for $\text{inClassIdTree}^*(\text{inClassId}(I_1), \ldots, \text{inClassId}(I_n))$, etc. (This abbreviation matches the grammar for product types in the next section.) Thus, for example, $\mathit{classOf}(\text{inObject}(I, r), d) = (I, \mathit{classOf}(d))$.

In the record model, messages cannot be sent to tuples, but in essence that is what multiple dispatch does in the generic function model. Thus, if a generic function is called with a tuple of arguments, it dispatches based on all the arguments. In our class-based generic function model, the dispatch is be based on a tuple of class names for the objects in the arguments. This contrasts with the record model, in which the method selected is based on the class of the first (implicit) argument only. Multiple dispatch has some expressiveness advantages

in practice, since the dispatch can be done symmetrically [11]. In particular, the generic function model helps solve part of the "binary method problem" [4].

Since generic functions are found in the second part of the environment, the syntax used with this model is typically chosen to match this semantics. That is, instead of writing $E_0.l(E_1)$, one writes $l(E_0, E_1)$. The semantics of the resulting client expressions is as follows.

$\mathcal{E}_g$: Expression $\rightarrow$ GEnvironment $\rightarrow$ Data$_\perp$
$\mathcal{E}_g[\![I]\!](\rho_d, \rho_f) = \rho_d(I)$
$\mathcal{E}_g[\![l(E)]\!]\ (\rho_d, \rho_f) = \rho_f(l)(classOf(\mathcal{E}_g[\![E]\!](\rho_d, \rho_f)))(\mathcal{E}_g[\![E]\!](\rho_d, \rho_f))$
$\mathcal{E}_g[\![(E_1, \ldots, E_n)]\!](\rho_d, \rho_f) = (\mathcal{E}_g[\![E_1]\!](\rho_d, \rho_f), \ldots, \mathcal{E}_g[\![E_n]\!](\rho_d, \rho_f))$

2.3 Comparing the Record and Generic Function Models

As one can see, the two kinds of models are similar, but there are two main differences.

- The way the second part of the environment is organized.
- In the generic function model, the method invoked depends, in general, on all of the arguments in the message, not just on the implicit argument.

We now discuss how to simulate each model with the other.

2.3.1 Simulating Generic Functions in the Record Model

Because of the second difference noted above, simulating the generic function model with the record model is not very elegant [4]. There are at least two ways to go about such a simulation, however.

One simulation simulates an n-ary generic function[1] that can handle k different types of arguments in each argument position by $k^{n+1} - k$ methods in the record model [4, 27]. For example, to simulate a binary generic function named `add` that works on the types `Int`, and `Float`, one would have 6 methods, as follows. (More explanation follows the code.)

```
class Int implements Number
  ...
  method add(o:Number): Number = o.addToInt(self)
  method addToInt(o:Int): Number = ...
  method addToFloat(o:Float): Number = ...
end

class Float implements Number
  ...
  method add(o: Number): Number = o.addToFloat(self)
  method addToInt(o: Int): Number = ...
```

[1] A generic function can be considered to be n-ary if it takes a tuple of n elements for its argument type.

```
  mehhod addToFloat(o: Float): Number = ...
end
```

In the coding above, we assume that `Number` is a common supertype of `Int` and `Float`. The two methods named `add`, found in objects of type `Int` and `Float`, dispatch to one of the four methods named `addToInt` or `addToFloat`. Each of these methods knows both its second argument's type and the type of `self`, so they can actually do the addition.

A second simulation of the generic function model by the record model uses objects that act like tuples of objects [4, Section 3.2]. In this simulation, one forms the argument tuple sent to a generic function object into a single object, whose type acts like the product of the argument types of the generic function. This has the disadvantage of using k^n such new types of objects, each with one method, to simulate an n-ary generic function that can handle k different argument types in each position. Still, this is fewer methods than needed by the first technique. This simulation also points out importance of dispatching on product types (tuples) in the generic function model.

2.3.2 Simulating the Record Model by the Generic Function Model

The simulation of the class-based record model by the class-based generic function model, while more straightforward and obvious in some ways, seems to be less well known. The basic idea is very simple. For each method in the record model of the form

```
class MyType
  ...
  method foo(x: T): S = E
end
```

one takes this method out of the class of definition, and adds the implicit argument to it, making it look as follows.

```
class MyType
  ...
end
method foo(self: MyType, x: T): S = E
```

The semantics groups these methods into generic functions.

Taking methods out of classes points out one problem with the generic function model, which is how to achieve information hiding. That can be solved by scoping [14], but the solution is outside the scope of this paper, since it does not concern client expressions.

We remind the reader that our simulation will only work for static, class-based languages. If we were not trying to simulate a static, class-based record model, we would have the following problems in working out the simulation in detail.

- How could one find all the methods with the same name?

  If the methods were only found in objects, we would have to find all objects reachable from a given environment, which is not computable in general. Having the classes be statically-known eliminates this problem.

- If objects contained methods, then two objects might differ only in their methods. Since the methods are not present in objects of the generic function model, how would such objects be distinguished by a generic function?

  Our use of a class-based model for objects avoids this problem, because the class of objects in the record model distinguishes two objects that would otherwise appear identical.

In summary, it seems that the generic function model cannot effectively simulate an arbitrary record model, but only one derived from a static, class-based language.

We now formalize the simulation of the static, class-based record model by the class-based generic function model in precise detail. This may make the simulation look more complex than it is, but we hope the details are instructive.

A translation from client expressions in the record model to those in the generic function model is given below by the function *toGF*. This translation is only defined for the client expressions we have been considering, but it could easily be extended to encompass additional client expressions such as **if**-expressions.

Definition 2.1 *Let E be a client expression in the record syntax. Then $toGF(\text{`}E\text{'})$ is an expression in the generic function syntax defined as follows.*

$$toGF(\text{`}I\text{'}) = I$$
$$toGF(\text{`}E_0.l(E_1)\text{'}) = l(toGF(\text{`}(E_0,E_1)\text{'}))$$
$$toGF(\text{`}(E_1,\ldots,E_n)\text{'}) = (toGF(\text{`}E_1\text{'}),\ldots,toGF(\text{`}E_n\text{'}))$$

For this translation to work, the environment must contain generic functions that simulate the methods found in the objects of the record model. The following defines a function that translates a record model's environment to an environment in the generic function model that can simulate it.

$$simRenv : \text{REnvironment} \to \text{GEnvironment}$$
$$simRenv(\rho_r, \rho_c) = (\rho_d, \rho_f)$$
$$\textbf{where } \rho_d = \rho_r$$
$$\rho_f = \{(l, gfFor(\rho_c, l)) \mid c \in range(\rho_c), l \in domain(c)\}$$

To construct a generic function for an environment and a label, we use the method from the first argument's class in the record model. This works because the domains of data are the same in the two models, and because we will only be passing to the record model's methods objects that come from it to begin with. We only deal with generic functions that take a pair of arguments, since

that is what the translation produces from calls to methods in the record model. Notice also that the dispatch ignores the second argument, since, after all, this is simulating single dispatch!

$gfFor : (\text{ClassId} \xrightarrow{\text{fin}} \text{RMethDict}) \times \text{Label} \to \text{GGenFun}$
$gfFor(\rho_c, l) = \lambda(I, ct)\ .\ \rho_c(I)(l)$

Once the simulating environment has been constructed, the following theorem holds.

Theorem 2.2 *Let $(\rho_r, \rho_c) \in REnvironment$ be an environment. Let E be a client expression in the record syntax. Then the following holds.*

$$\mathcal{E}_g[\![toGF(\text{`}E\text{'})]\!](simRenv(\rho_r, \rho_c)) = \mathcal{E}_r[\![E]\!](\rho_r, \rho_c)$$

Proof: (By structural induction on E.)
Let (ρ_d, ρ_f) be defined as follows.

$$(\rho_d, \rho_f) = simRenv(\rho_r, \rho_c) \tag{1}$$

Then by definition of *simRenv*, the following hold.

$$\rho_d(I) = \rho_r(I) \tag{2}$$
$$\rho_f(l) = gfFor(\rho_c, l) \tag{3}$$

For the base case, suppose E is an identifier, I. We calculate as follows.

$\mathcal{E}_g[\![toGF(\text{`}I\text{'})]\!](simRenv(\rho_r, \rho_c))$
$=$ $\langle$by definition of $toGF$, equation (1)$\rangle$
$\mathcal{E}_g[\![I]\!](\rho_d, \rho_f)$
$=$ $\langle$by definition of $\mathcal{E}_g\rangle$
$\rho_d(I)$
$=$ $\langle$by equation (2)$\rangle$
$\rho_r(I)$
$=$ $\langle$by definition of $\mathcal{E}_r\rangle$
$\mathcal{E}_r[\![I]\!](\rho_r, \rho_c)$

For the inductive cases, the inductive hypothesis is that the result holds for each subexpression.

Suppose E is of the form $E_0.l(E_1)$. We calculate as follows.

$\mathcal{E}_g[\![toGF(\text{`}E_0.l(E_1)\text{'})]\!](simRenv(\rho_r))$
$=$ $\langle$by definition of $toGF$, equation (1)$\rangle$
$\mathcal{E}_g[\![l(toGF(\text{`}(E_0, E_1)\text{'}))]\!](\rho_d, \rho_f)$
$=$ $\langle$by definition of $\mathcal{E}_g\rangle$
$\rho_f(l)(classOf(\mathcal{E}_g[\![toGF(\text{`}(E_0, E_1)\text{'})]\!](\rho_d, \rho_f)))$
$\quad(\mathcal{E}_g[\![toGF(\text{`}(E_0, E_1)\text{'})]\!](\rho_d, \rho_f))$
$=$ $\langle$by the inductive hypothesis$\rangle$

$\rho_f(l)(classOf(\mathcal{E}_r[\![(E_0, E_1)]\!](\rho_r, \rho_c)))(\mathcal{E}_r[\![(E_0, E_1)]\!](\rho_r, \rho_c))$

$=$ $\langle$by equation (3)$\rangle$

$gfFor(\rho_c, l)(classOf(\mathcal{E}_r[\![(E_0, E_1)]\!](\rho_d, \rho_f)))(\mathcal{E}_r[\![(E_0, E_1)]\!](\rho_d, \rho_f))$

$=$ $\langle$by the definition of $\mathcal{E}_r\rangle$

$gfFor(\rho_c, l)(classOf(\mathcal{E}_r[\![E_0]\!](\rho_r, \rho_c), \mathcal{E}_r[\![E_1]\!](\rho_r, \rho_c)))$
$(\mathcal{E}_r[\![E_0]\!](\rho_r, \rho_c), \mathcal{E}_r[\![E_1]\!](\rho_r, \rho_c))$

Now there are two cases.

If isObject$(\mathcal{E}_r[\![E_0]\!](\rho_r, \rho_c))$ is false, then the last formula in the calculation above is $\perp$. But in this case, $\mathcal{E}_r[\![E_0.l(E_1)]\!](\rho_r, \rho_c)$ is also $\perp$, by definition of $\mathcal{E}_r$. So the result holds in this case.

In the second case, $\mathcal{E}_r[\![E_0]\!](\rho_r, \rho_c)$ is in the Object summand of Data. So we can make the following abbreviations corresponding to the bindings introduced in the definition of *gfFor*.

$$d_0 = \mathcal{E}_r[\![E_0]\!](\rho_r, \rho_c) \quad (4)$$

$$d_1 = \mathcal{E}_r[\![E_1]\!](\rho_r, \rho_c) \quad (5)$$

Also let (I, r_0) be the object in d_0. That is the following holds.

$$d_0 = \text{inObject}(I, r_0) \quad (6)$$

Now we continue our calculation.

$\mathcal{E}_g[\![toGF(\text{`}E_0.l(E_1)\text{'})]\!](simRenv(\rho_r, \rho_c))$

$=$ $\langle$by the previous calculation$\rangle$

$gfFor(\rho_c, l)(classOf(\mathcal{E}_r[\![E_0]\!](\rho_r, \rho_c), \mathcal{E}_r[\![E_1]\!](\rho_r, \rho_c)))$
$(\mathcal{E}_r[\![E_0]\!](\rho_r, \rho_c), \mathcal{E}_r[\![E_1]\!](\rho_r, \rho_c))$

$=$ $\langle$by the abbreviations above for d_0 and $d_1\rangle$

$gfFor(\rho_c, l)(classOf(d_0, d_1))(d_0, d_1)$

$=$ $\langle$by equation (6)$\rangle$

$gfFor(\rho_c, l)(classOf(\text{inObject}(I, r_0), d_1))(\text{inObject}(I, r_0), d_1)$

$=$ $\langle$by definition of *classOf* (using our notational abbreviation)$\rangle$

$gfFor(\rho_c, l)(I, classOf(d_1))(\text{inObject}(I, r_0), d_1)$

$=$ $\langle$by definition of *gfFor*$\rangle$

$\rho_c(I)(l)(\text{inObject}(I, r_0), d_1)$

$=$ $\langle$by definition of $\mathcal{E}_r$ and the abbreviations above$\rangle$

$\mathcal{E}_r[\![E_0.l(E_1)]\!](\rho_r, \rho_c)$

This completes the case where E is a message send expression.

Suppose E is of the form $(E_1, \ldots, E_n)$, where $n \geq 0$. We calculate as follows.

$\mathcal{E}_g[\![toGF(\text{`}(E_1, \ldots, E_n)\text{'})]\!](simRenv(\rho_r, \rho_c))$

$=$ $\langle$by definition of *toGF*$\rangle$

$\mathcal{E}_g[\![(toGF(\text{`}E_1\text{'}), \ldots, toGF(\text{`}E_n\text{'}))]\!](simRenv(\rho_r, \rho_c))$

$=$ $\langle$by definition of $\mathcal{E}_g\rangle$

$(\mathcal{E}_g[\![toGF(\text{`}E_1\text{'})]\!](simRenv(\rho_r, \rho_c)), \ldots, \mathcal{E}_g[\![toGF(\text{`}E_n\text{'})]\!](simRenv(\rho_r, \rho_c)))$

$=$ $\langle$by the inductive hypothesis$\rangle$

$$
\begin{array}{ll}
 & (\mathcal{E}_r[\![E_1]\!](\rho_r,\rho_c),\ldots,\mathcal{E}_r[\![E_n]\!](\rho_r,\rho_c)) \\
= & \langle\text{by definition of } \mathcal{E}_r\rangle \\
 & (\mathcal{E}_r[\![(E_1,\ldots,E_n)]\!](\rho_r,\rho_c)
\end{array}
$$

∎

Note that this simulation does not affect integers and booleans, hence the translation $toGF$ preserves observable outputs (i.e., integer and boolean results) from client expressions.

2.4 Discussion

The main conclusions from the simulations described above are the following. First, it seems to take exponentially many objects or methods for the record model to simulate the generic function model. Second, the generic function model can only simulate a class-based record model with statically-known classes.

Since the class-based generic function and record models can each simulate each other, in some narrow, technical sense they are equivalent. However, as a practical matter, the ability of one model to simulate the other is not the same thing as ease of programming a simulation of one model in the other.

While we spent more effort on the simulation of the record model with the generic function model, that simulation requires less programming effort, since it simply rearranges the information present in the record model in a way that is standard when programming in the generic function model.

On the one hand, the known simulations of the generic function model by the record model are not as easy, and engender an explosion in either the number of methods or the number of objects [4]. Hence, as a practical programming matter, one can fairly say that multiple-dispatching languages are more flexible (expressive) than single-dispatching languages.

On the other hand, there is some expressive power gained by the general case of the record model, one that is not based on classes, that seems difficult for the generic function model to simulate. But the most popular single-dispatching languages, C++, Java, Eiffel, and Smalltalk, are all class-based, so this expressive power gain may not be that important in practice.

Our theoretical modeling activity has been based on abstractions of the class-based generic function model. The idea is that it is easy to take a particular instance of a static, class-based record model and transform it to such a model, and in that setting use our results. Since reasoning about programs is a static activity, the assumption that the set of classes is statically-known seems like a small limitation.

3 Typing and Subtyping

The purpose of a type system is to enforce data abstraction and to prevent obviously incorrect programs. From the point of view of client code, we can enforce data abstraction in an OO language by not allowing clients direct access

to data fields in objects. Obviously incorrect programs can be prevented if the type system prevents sending messages that will not find a method or that have the wrong number of arguments. (The type system should also prevent looking up identifiers that are not in the environment.)

In the following we discuss type systems that are appropriate to the record and generic function models, and compare them.

3.1 Types in the Record Model

We use the following as the abstract syntax of type attributes for the class-based record model. (As might be guessed from the names, the DataType and MethodType attributes will be the same in the generic function model.)

$RT, T, S, U \in$ RecordModelType
$D \in$ DataType
$M \in$ MethodType
$RMD \in$ RecordMethDictType
$I \in$ Identifier
$l \in$ Label

$T ::= D \mid M \mid RMD$
$RD ::= \text{int} \mid \text{bool} \mid I \mid (D_1, \ldots, D_n)$ where $n \geq 0$
$M ::= D_1 \to D_2$
$RMD ::= [l_1 : M_1, \ldots, l_n : M_n]$ where $n \geq 0$

Order of the bindings in a method dictionary type does not matter, and duplicate labels are not allowed. We will sometimes abbreviate product types using vector notation, such as $\vec{D}$.

To handle recursive types, such as the types of methods that return `self`, the type environment will associate class names to method dictionary types. This is similar to using recursive type binders (μ types) [1, Chapter 9].

Following the semantics, type environments in the record model consist of a pair of finite functions.

$$(\pi_r, \pi_c) \in \text{RTypeEnv} = (\text{Identifier} \xrightarrow{\text{fin}} \text{DataType}) \times (\text{ClassId} \xrightarrow{\text{fin}} \text{RecordMethDictType})$$

A type U is a *subtype* of T, written $U \leq T$, and T is a *supertype* of U, if data of type U can be used in place of data of type T without type error. For object types, this means that every message that can be sent to a T object can also be sent to a U object. A message with name l_i can be sent to an object o if o's class is bound in the class part (π_c) of the type environment to a method dictionary type, and if that method dictionary type binds the label l_i to a method type, $D_i \to D_i'$, such that the type of the argument is D_i. Hence, U must have all the methods of T, and perhaps some extra methods.

Cardelli was the first to propose a sound type system that can statically determine subtyping for the function, and immutable record and variant types [7]. We adapt Cardelli's algorithm for deciding subtype relationships to our situation in the following inference rules. These rules are given with respect to the class part of the type environment, π_c, which maps class names to method dictionary types. This reflects the type system's static knowledge about classes. Note, however, that, although class names are used, subtyping is decided structurally. In effect, this is very similar to the use of a recursive type binder (usually written μ [1, Chapter 9]).

(As usual, the hypotheses are above the horizontal line, the conclusion below, the rule name in square brackets to the left, and side conditions to the right. Judgements of the form $\pi_c \vdash S \leq T$ mean that one can prove that S is a subtype of T.)

[refl] $$\pi_c \vdash T \leq T$$

[tran] $$\frac{\pi_c \vdash S \leq U, \pi_c \vdash U \leq T}{\pi_c \vdash S \leq T}$$

[id-1] $$\frac{\pi_c \vdash S \leq T}{\pi_c \vdash I \leq T} \quad \text{if } (I, S) \in \pi_c$$

[id-2] $$\frac{\pi_c \vdash T \leq S}{\pi_c \vdash T \leq I} \quad \text{if } (I, S) \in \pi_c$$

[fun] $$\frac{\pi_c \vdash T' \leq T, \vdash S \leq S'}{\pi_c \vdash (T \to S) \leq (T' \to S')}$$

[prod] $$\frac{\pi_c \vdash T_1 \leq T_1', \ldots, \pi_c \vdash T_n \leq T_n'}{\pi_c \vdash (T_1, \ldots, T_n) \leq (T_1', \ldots, T_n')} \quad \text{if } n \geq 0$$

[rmd] $$\frac{\pi_c \vdash T_1 \leq T_1', \ldots, \pi_c \vdash T_n \leq T_n'}{\pi_c \vdash [l_1\colon T_1, \ldots, l_n\colon T_n, l_{n+1}\colon T_{n+1}, \ldots, l_m\colon T_m] \leq [l_1\colon T_1', \ldots, l_n\colon T_n']} \quad \text{if } 0 \leq n \leq m$$

The rule for function types [fun] is called the "contravariant rule." One consequence of this rule is that if a method dictionary type RMD is a subtype of RMD', then the argument types of the common methods in RMD must be supertypes of their types in RMD' [8, 17, 1].

We define the "domain" of a method dictionary type as follows.

$$domain([l_1 : M_1, \ldots, l_n : M_n]) = \{l_1, \ldots, l_n\} \tag{7}$$

As one goes up the subtyping lattice of method dictionary types, the domains can only shrink, not expand.

Lemma 3.1 *Let RMD and RMD' be elements of RecordMethDictType. If $\pi_c \vdash RMD \leq RMD'$ then $domain(RMD') \subseteq domain(RMD)$.* ∎

Type checking client expressions with respect to a type environment boils down to the following rules, which are again adapted from Cardelli's rules [1, 7]. (As usual, the type environment, written (π_r, π_c), is written to the left of the turnstile ($\vdash$) in judgements that expressions have a given type.)

$$\text{[id]} \quad (\pi_r, \pi_c) \vdash I : T \quad \text{if } \pi_r(I) = T$$

$$\text{[msg]} \quad \frac{\pi_c \vdash T_0 \leq [l : T_1 \to S], \ (\pi_r, \pi_c) \vdash E_0 : T_0, \ (\pi_r, \pi_c) \vdash E_1 : T_1}{(\pi_r, \pi_c) \vdash E_0.l(E_1) : S}$$

$$\text{[tup]} \quad \frac{(\pi_r, \pi_c) \vdash E_1 : T_1, \ \ldots, \ (\pi_r, \pi_c) \vdash E_n : T_n}{(\pi_r, \pi_c) \vdash (E_1, \ldots, E_n) : (T_1, \ldots, T_n)} \quad \text{if } n \geq 0$$

In the [msg] rule, we depend on the subtyping rules to massage the type of E_0 instead of using a subsumption rule (as does Cardelli [1, 7]), but this detail has no great weight.

We note the following facts about this proof system for later use. Its proof is an easy consequence of the definitions.

Lemma 3.2 *If* $(\pi_r, \pi_c) \vdash E : T$ *then* $T \in DataType$. ∎

3.2 Types in the Generic Function Model

We use the following as the abstract syntax of type attributes for the generic function model. We repeat the definitions of the DataType and MethodType attributes for convenience.

$GT, T, S, U \in$ GFModelType
$D \in$ DataType
$M \in$ MethodType
$GF \in$ GenericFunctionType
$I, l \in$ Identifier

$T ::= D \mid M \mid GF$
$D ::= \text{int} \mid \text{bool} \mid I \mid (D_1, \ldots, D_n)$ where $n \geq 0$
$M ::= D_1 \to D_2$
$GF ::= \{M_1, \ldots, M_n\}$ where $n \geq 0$

The order of the method types in $\{M_1, \ldots, M_n\}$ does not matter, and the argument types of each of the M_i must all be pairwise distinct. (See also [9] for a monotonicity requirement on such types that we are postponing discussing until later.)

Since there are no method dictionaries in the generic function model, it is not immediately obvious how to do decide subtype relationships structurally. Recall that, in the record model, class names could be mapped to a method dictionary type, but it is not clear what the analogous information would be that would allow structural type decisions to be made about subtyping and type checking. For this reason, and also to promote information hiding, generic function languages often feature by-name type checking. This is the case, for example, in Cecil [13], and in the theoretical work of Castagna *et al.* [9, 10].

Since there is no structural information about class names, the rules for determining subtype relationships rely on an assumed subtype ordering on atomic (i.e., non-tuple) class names. This assumed ordering is given the name A in the rules below; it is a preorder on type names. We assume that the base types `int` and `bool` are in the domain of A, but are not related to any other names by A. Class names that are tuples are handled by the rules below.

The following rules determining subtyping relationships in this model. The rule [gf-g] is from the $\lambda\&$-calculus [9, Page 46][10].

[refl-g]	$A \vdash T \leq T$	
[tran-g]	$\dfrac{A \vdash S \leq U, A \vdash U \leq T}{A \vdash S \leq T}$	
[base-g]	$A \vdash S \leq T$	if $(S,T) \in A$
[fun-g]	$\dfrac{A \vdash T' \leq T, \vdash S \leq S'}{A \vdash (T \to S) \leq (T' \to S')}$	
[prod-g]	$\dfrac{A \vdash T_1 \leq T_1', \ldots, A \vdash T_n \leq T_n'}{A \vdash (T_1, \ldots, T_n) \leq (T_1', \ldots, T_n')}$	if $n \geq 0$
[gf-g]	$\dfrac{\forall 1 \leq i \leq n.\, \exists 1 \leq j \leq m.\, A \vdash S_j \leq T_i}{A \vdash \{S_1, \ldots, S_m\} \leq \{T_1, \ldots, T_n\}}$	if $n \geq 0$, $m \geq 0$

Following the semantics, type environments in the generic function model consist of a pair of finite functions.

$$(\pi_d, \pi_f) \in \text{GTypeEnv} = (\text{Identifier} \xrightarrow{\text{fin}} \text{DataType}) \times (\text{Label} \xrightarrow{\text{fin}} \text{GenericFunctionType})$$

In the typing rules below, judgements of the form $A; (\pi_d, \pi_f) \vdash E\colon T$ mean that assuming the atomic subtyping relationships in A and the typings in (π_d, π_f), the expression E has type T.

[id-g]	$A; (\pi_d, \pi_f) \vdash I\colon T$	if $\pi_d(I) = T$
[msg-g]	$\dfrac{A \vdash U \leq \{T \to S\},\ A; (\pi_d, \pi_f) \vdash E\colon T}{A; (\pi_d, \pi_f) \vdash l(E)\colon S}$	if $\pi_f(l) = U$
[tup-g]	$\dfrac{A; (\pi_d, \pi_f) \vdash E_1\colon T_1,\ \ldots,\ A; (\pi_d, \pi_f) \vdash E_n\colon T_n}{A; (\pi_d, \pi_f) \vdash (E_1, \ldots, E_n)\colon (T_1, \ldots, T_n)}$	if $n \geq 0$

3.3 Comparing Types in the two Models

Informally, several facets of the two type systems stand out in comparison.

- Type checking is by-name in the generic function model, hence the differences in assumptions of the subtyping rules and the replacement of the [id-1] and [id-2] subtyping rules by the [base-g] rule.
- Because of the difference in the organization of method types, the [rmd] rule is replaced by the [gf-g] rule.

One way to compare the [rmd] rule and the [gf-g] rule is to derive from the [gf-g] rule one similar in format to the [rmd] rule. The rule we have in mind is the following, which, as one can see by comparison, has a striking similarity to the [rmd] rule.

$$\text{[rmd-g]} \quad \frac{A \vdash T_1 \leq T_1', \ldots, A \vdash T_n \leq T_n'}{A \vdash \{T_1, \ldots, T_n, T_{n+1}, \ldots, T_m\} \leq \{T_1', \ldots, T_n'\}} \quad \text{if } 0 \leq n \leq m$$

The [rmd-g] rule can be derived from the [gf-g] rule as follows. Suppose the hypotheses of the [rmd-g] rule hold. Then for each $1 \leq i \leq n$, there is some $1 \leq j \leq m$, namely i, since $i \leq n \leq m$, such that $T_j \leq T_i'$, since $T_i = T_j$ and $T_i \leq T_j$. This fulfilling the hypothesis of the [gf-g] rule, so by the [gf-g] rule the conclusion of the [rmd-g] rule follows.

We now propose to show how the simulation of the record model by the generic function model carries over into the typings. What we are aiming at is a theorem that says that if an expression type checks in the record model, then the translation of that expressions (using $toGF$) has the translated type in the generic function model.

In order to bridge the gap between the by-name subtyping in the generic function model and the structural subtyping in the record model, we first need to construct the set A of subtype relationships among class names that is required. This is done by extracting all such subtyping relationships from the record model's type rules, and forming them into a binary relation, which will be reflexive and transitive by definition of the subtyping rules.

$$\begin{aligned}
&\mathit{atomicSubs}(\pi_c) : (\text{ClassId} \xrightarrow{\text{fin}} \text{RecordMethDictType}) \to (\text{ClassId} \times \text{ClassId}) \\
&\mathit{atomicSubs}(\pi_c) = \{(I_1, I_2) \mid \pi_c \vdash I_1 \leq I_2\}
\end{aligned}$$

Second, we have to construct the type environment needed by the generic function model from the record model's type environment. The data part can be used unchanged, and we explain how to construct the generic function types from the class part below.

$$\begin{aligned}
&\mathit{simTenv} : \text{RTypeEnv} \to \text{GTypeEnv} \\
&\mathit{simTenv}(\pi_r, \pi_c) = (\pi_d, \pi_f) \\
&\quad \textbf{where } \pi_d = \pi_r \\
&\qquad\quad \pi_f = \{(l, \mathit{gfTfor}(l, \pi_c)) \mid RMD \in \mathit{range}(\pi_c), l \in \mathit{domain}(RMD)\}
\end{aligned}$$

To construct a generic function type for a type environment and a label, we collect the types of all the methods in the type environment with that label. This works because the type attributes for methods are the same in both models. This translation only produces generic function types that take a pair of arguments, the first of which is the type of the record model's `self` parameter.

$$\mathit{gfTfor} : (\text{ClassId} \xrightarrow{\text{fin}} \text{RecordMethDictType}) \times \text{Label} \to \text{GenericFunctionType}$$

$$gfTfor(\pi_c, l) = \{(I, D_1) \to D_2 \mid I \in domain(\pi_c), (l : D_1 \to D_2) \in \pi_c(I)\}$$

Once the simulating type environment has been constructed, the following theorem holds.

Theorem 3.3 *Let $(\pi_r, \pi_c) \in RTypeEnv$ be a type environment. Let E be a client expression in the record syntax. Let $T \in DataType$ be a type. If $(\pi_r, \pi_c) \vdash E : T$ in the record model, then in the generic function model*

$$atomicSubs(\pi_c); simTenv(\pi_r, \pi_c) \vdash toGF(\text{`}E\text{'}) : T.$$

Proof: Suppose $(\pi_r, \pi_c) \vdash E : T$ in the record model.
Let A and (pi_d, π_f) be as follows.

$$A = atomicSubs(\pi_c) \tag{8}$$
$$(\pi_d, \pi_f) = simTenv(\pi_r, \pi_c) \tag{9}$$

By definition of *simTenv* the following holds.

$$\pi_d = \pi_c \tag{10}$$

We proceed by induction on the structure of the proof of this typing.

For the base case, E is an identifier, I. Then by the proof rule [id], $\pi_r(I) = T$. Thus since $\pi_d = \pi_c$, we have $\pi_d(I) = T$. Since $toGF(\text{`}I\text{'}) = I$, the conclusion follows from the proof rule [id-g].

For the inductive cases, the inductive hypothesis is that the result holds for each subexpression.

Suppose E is of the form $E_0.l(E_1)$. Then by the proof rule [msg], there are types T_0, T_1, and S such that

$$\pi_c \vdash T_0 \leq [l : T_1 \to S], \tag{11}$$
$$(\pi_r, \pi_c) \vdash E_0 : T_0, \tag{12}$$
$$(\pi_r, \pi_c) \vdash E_1 : T_1. \tag{13}$$

In this case $toGF(\text{`}E_0.l(E_1)\text{'}) = l(E_0, E_1)$. By the inductive hypothesis we have the following.

$$A; (\pi_d, \pi_f) \vdash (E_0, E_1) : (T_0, T_1) \tag{14}$$

We can finish the proof in this case by using the [msg-g] typing rule, if we can show that $\pi_f(l)$ is defined and $A \vdash \pi_f(l) \leq \{(T_0, T_1) \to S\}$.

To show that $\pi_f(l)$ is defined, first note that equation (11) makes T_0 a subtype of a method dictionary type with label l in its domain. Since T_0 is the type of an expression, it is an element of DataType by Lemma 3.2. But since T_0 is a subtype of a method dictionary type and an element of DataType, by the record model subtyping rules T_0 must be an identifier that is in the domain of π_c. Furthermore, by the subtyping rules, $\pi_c(T_0)$ must be a subtype of $[l : T_1 \to S]$. By Lemma 3.1, $\pi_c(T_0)$ must have label l in its domain. Hence by construction of *simTenv*, $\pi_f(l)$ is defined.

Since $\pi_f(l)$ is defined, by definition of *simTenv*, $\pi_f(l)$ is *gfTfor*(l, π_c). So by definition of *gfTfor*, there are types D_1 and D_2 such that $(l : D_1 \to D_2) \in \pi_c(T_0)$ and $\pi_f(l)$ contains the method type $(T_0, D_1) \to D_2$. By the subtyping rules for the record model, this means that

$$\pi_c \vdash (D_1 \to D_2) \leq (T_1 \to S). \tag{15}$$

Thus by the subtyping rule [fun], we have

$$\pi_c \vdash T_1 \leq D_1, \tag{16}$$
$$\pi_c \vdash D_2 \leq S. \tag{17}$$

Since D_1 and D_2 are elements of DataType, by definition of *atomicSubs* we have in the generic function world

$$A \vdash T_1 \leq D_1, \tag{18}$$
$$A \vdash D_2 \leq S. \tag{19}$$

It follows by the rules [refl-g] and [prod-g] that

$$A \vdash (T_0, T_1) \leq (T_0, D_1). \tag{20}$$

Now using [fun-g] we have that

$$A \vdash ((T_0, D_1) \to D_2) \leq ((T_0, T_1) \to S). \tag{21}$$

So by the [gf-g] rule, we have that

$$A \vdash \{(T_0, D_1) \to D_2\} \leq \{(T_0, T_1) \to S\}. \tag{22}$$

Since $\pi_f(l)$ contains the method type $(T_0, D_1) \to D_2$, it follows by the [gf-g] (or [rmd-g]) rule that

$$A \vdash \pi_f(l) \leq \{(T_0, T_1) \to S\}. \tag{23}$$

This completes the case for message send expressions.

Suppose E is of the form $(E_1, \ldots, E_n)$, where $n \geq 0$. Then the result follows directly from the inductive hypothesis. ∎

3.4 Monotonicity

So far we have ignored implementation-side type checking questions in our treatment of the generic function model. These questions determine whether generic function types are well formed in the sense that a call to a generic function will be able to select a unique most-specific method [9, 10, 13]. Such considerations have not concerned us, since we have only worried about type checking for client expressions, and have assumed that appropriate methods were reflected in the type of a generic function.

However, there is at least one property of generic function types as a whole that is a concern for client-side checking. This is the monotonicity property [45],

which says that as more information is known about the values of expressions, the type of the expression does not become larger, but can only become a subtype of the type originally inferred for it [9, 10]. In terms of an operational semantics, this says that if E reduces to E', then the type of E' must be a subtype of the type of E. Clearly such a property is necessary for type soundness in a type system that regards static types as upper bounds.

One way to describe this condition is to require that in each generic function type $\{D_1 \to D'_1, \ldots, D_n \to D'_n\}$, whenever $A \vdash D_i \leq D_j$ then $D'_i \leq D'_j$ [9, Page 45]. We shall see another way of describing this condition in the next section.

In our simulation of the record model by the generic function model, does the type environment constructed for the generic function model only contain types that satisfy the monotonicity condition? Yes, as shown by the following.

Lemma 3.4 *Let $(\pi_d, \pi_c) \in RTypeEnv$ be a type environment. Let l be a label. Then the generic function type $gfTfor(\pi_c, l)$ satisfies the monotonicity condition.*

Proof: Let $A = atomicSubs(\pi_c)$. Without loss of generality, suppose that

$$\{I_1, \ldots, I_n\} = \{I \mid l \in domain(\pi_c(I))\}. \tag{24}$$

Further, for each $1 \leq i \leq n$ let D_i and D'_i be defined by

$$(l : D_i \to D'_i) \in \pi_c(I_i) \tag{25}$$

Then we have

$$gfTfor(\pi_c, l) = \{(I_1, D_1) \to D'_1, \ldots, (I_n, D_n) \to D'_n\} \tag{26}$$

Suppose for some $1 \leq i \leq j \leq n$,

$$A \vdash (I_i, D_i) \leq (I_j, D_j). \tag{27}$$

It follows from [prod-g] that

$$A \vdash I_i \leq I_j \tag{28}$$

$$A \vdash D_i \leq D_j \tag{29}$$

By the formula (28), it follows that $\pi_c \vdash \pi_c(I_i) \leq \pi_c(I_i)$. Since these are method dictionary types, by the looking at how the [rmd] rule affects the method for the label l, we have

$$\pi_c \vdash (D_i \to D'_i) \leq (D_j \to D'_j). \tag{30}$$

But by the [fun] rule, we obtain the desired result.

$$\pi_c \vdash D'_i \leq D'_j \tag{31}$$

∎

3.5 Related Work

Many authors have studied the semantics and typing of OO languages. Few have studied the relationship between the single-dispatching and multiple-dispatching languages in detail.

One notable exception is Castagna. In his recent book [9], Castagna treats the theory of generic function languages in detail. Chapter 3 of that book is comparable to what we have done so far, in that Castagna treats a static, class-based singly-dispatched language (KOOL) and compares it to a language with CLOS-style generic functions (CBL). He shows how to add encapsulated multi-methods (see Section 3.1.11 and [4]) to KOOL. Unlike our models, the languages Castagna treats are full languages, and he thus compares aspects of the implementation of objects that we ignore. The comparison, however between the two languages is informal.

4 Algebraic Models

The class-based generic function model described above has several details that are inessential from the point of view of client code:

- The data in an object is accessed through several named fields.
- Generic functions map class names (and tuples of names) to methods.

While these details correspond to implementations of languages like CLOS, Dylan, and Cecil, they are not directly relevant for reasoning about client code. Client code has no direct access to fields by definition. Furthermore, client code can only call methods through generic functions; it has no direct access to methods either.

Therefore, it is helpful to take an additional step of abstraction. This is especially true if one is concerned with how to reason about client code that uses objects, as opposed to reasoning about implementations of objects. This abstraction step takes one from the generic function models described above to various algebraic models.

4.1 Signatures

Another way to capture the monotonicity requirement for generic function types is found in the work of Reynolds [45, 46] and of Goguen and Meseguer [22, 23] on algebraic models. In the tradition of universal algebra, this work collects the type information into a mathematical structure. We call this structure a "signature with subtyping" (to distinguish it from signatures without subtypes).

A signature with subtyping consists of type names, the assumed preorder on type names (now just written $\leq$), and the type information for generic functions. The type information stored for generic functions is an abstraction of the type attributes noted earlier. The important information is the mapping from argument types to result types, which is needed to check calls to generic

functions. This mapping is represented directly in the signature by the *ResType* mapping. Doing this allows the crucial requirement of monotonicity to be stated succinctly.

Definition 4.1 *A* signature with subtyping, $\Sigma = (TYPE, \leq, OP, ResType)$, *consists of:*

- *a nonempty set TYPE of* type names,
- *a preorder*[2] $\leq$ *on TYPE, and by pointwise extension on* $TYPE^*$,
- *a set OP of* operation symbols, *and*
- *a partial function,* $ResType: OP \times TYPE^* \rightarrow TYPE_\perp$ *that is monotonic in the following sense. Whenever* $ResType(g, \vec{T})$ *is defined and* $\vec{U} \leq \vec{T}$, *then* $ResType(g, \vec{U})$ *is defined and* $ResType(g, \vec{U}) \leq ResType(g, \vec{T})$.

Note that the set of types is no longer closed under the formation of product types, but in this respect we follow the algebraic tradition of "flat" argument lists.

The above definition essentially follows Reynolds [45, Pages 217–218]. The set *OP* is what we called "Label" in the two models of objects discussed above.

In Goguen and Meseguer's work on order sorted algebra [23] [22, pp. 8–9], there is a "monotonicity" condition on signatures that has the same effect as the monotonicity condition on *ResType* above. Their "regularity" condition on signatures has the effect of allowing the result type of an operator to be given as a function of the arguments. Hence Goguen and Meseguer's definition of a signature with subtyping is essentially equivalent to the one we use by Reynolds.

4.2 Subtype Polymorphic Algebras

It is traditional in algebraic models to ignore internal structure in data; this is the main idea behind the algebraic approach to specification [20, 21, 26]. The standard mathematical structure used in universal algebra, an algebra, is an abstraction of the code used in an OO program. We will call our variant that takes subtyping into account a "subtype polymorphic algebra."

Definition 4.2 *Let* $\Sigma = (TYPE, \leq, OP, ResType)$ *be a signature with subtyping. A* subtype polymorphic Σ-algebra, $\boldsymbol{A} = (A, OP^{\boldsymbol{A}})$, *consists of:*

- *a family of sets* $A = \langle A_T : T \in TYPE \rangle$ *called the* carrier *of* $\boldsymbol{A}$, *where* A_T *is the carrier set of the type* T, *and*
- *a set of* operation interpretations, $OP^{\boldsymbol{A}} = \{g^{\boldsymbol{A}} : g \in OP\}$, *where for each* $g \in OP$, $g^{\boldsymbol{A}}$ *is a partial function of type* $A^* \rightarrow A_\perp$ *that agrees with the signature* Σ *in the following sense. Whenever* $ResType(g, \vec{T})$ *is defined and* $\vec{o} \in A_{\vec{T}}$, *then* $g^{\boldsymbol{A}}(\vec{o}) \in \bigcup_{U \leq ResType(g, \vec{T})} A_U$.

[2] A *preorder* is a reflexive and transitive binary relation.

A type T's carrier set, A_T, is just a set; it models the values of objects of type T. Sending a message named l is interpreted by calling the operation interpretation $l^{\mathbf{A}}$, which abstracts away the details of looking up l in the generic function model's environment, using the arguments to find a method, and then passing these arguments to the method.

The agreement condition on operation interpretations can be stated more simply by defining an abbreviation for the union of all carrier sets of types below a given type in the subtype ordering. We define this as follows.

$$\hat{A}_T \stackrel{\text{def}}{=} \bigcup_{U \leq T} A_U \tag{32}$$

With this definition, we can state that an operation interpretation must be such that if $ResType(g, \vec{T}) = U$ and $\vec{o} \in A_{\vec{T}}$, then

$$g^{\mathbf{A}}(\vec{o}) \in \hat{A}_U. \tag{33}$$

The following constraint on the structure of subtype polymorphic algebras follows trivially from the agreement condition. It is useful as something to keep in mind about subtype polymorphic algebras.

Corollary 4.3 *Let $\Sigma = (TYPE, \leq, OP, ResType)$ be a signature with subtyping. Let $\mathbf{A}$ be a subtype polymorphic Σ-algebra . Then for each $g \in OP$, for $U, U' \in TYPE$, and for $\vec{S}, \vec{T} \in TYPE^*$, if $ResType(g, \vec{S}) = U$, $ResType(g, \vec{T}) = U'$, and $\vec{o} \in A_{\vec{S}} \cap A_{\vec{T}}$, then $g(\vec{o}) \in \hat{A}_U \cap \hat{A}_{U'}$.* ∎

There are two variations on subtype polymorphic algebras that are important for modeling OO programs. These variations are in how the carrier sets of subtypes are related to the carrier sets of supertypes. In the first kind of model, the carrier sets of subtypes are subsets of the carrier sets of their supertypes' carrier sets. In the second the carrier sets are expected to be disjoint. Although these variations primarily concern the carrier sets, they also affect the mathematics of the operation interpretations.

4.3 Order-Sorted Algebras

Goguen and Meseguer's *order-sorted algebras* were originally designed to solve expressiveness problems in algebraic specification [22, 23]. However, they can also be seen as an abstraction of generic function languages in which subtypes are required to be subsets [6, 41].

In order-sorted algebras, the carrier sets of subtypes must be subsets of the carrier sets of their supertypes. Another way of putting this, used in the definition below, is that a supertype's carrier set is the union of the carrier sets of its subtypes. The motivation for this condition is that it allows operation interpretations to be very simple — they are just functions. An operation interpretation $g^{\mathbf{A}}$ works on subtypes because functions work on all subsets of their domains. Hence subtype polymorphism is modeled in a natural way, without any additional mathematical complications.

Definition 4.4 *Let* $\Sigma = (TYPE, \leq, OP, ResType)$ *be a signature with subtyping. A subtype polymorphic* Σ*-algebra* $\boldsymbol{A}$ *is a* order-sorted Σ-algebra *if and only if for each* $T \in TYPE$, $A_T = \hat{A}_T$.

Our definition of order-sorted algebras differs from Goguen and Meseguer's [22, Page 10] only in that they do not use polymorphic operation interpretations. Instead, they index operation symbols by their types. This is equivalent to writing operation symbols with subscripts, such as $g_{\vec{S},U'}$. Goguen and Meseguer give interpretations symbols separately for each index. In doing so, they are obligated to state an additional monotonicity condition on the operation interpretations $g^{\boldsymbol{A}}_{\vec{S},U'}$ and $g^{\boldsymbol{A}}_{\vec{T},U}$ when $\vec{S} \leq \vec{T}$. This condition is that $g^{\boldsymbol{A}}_{\vec{S},U'}$ equals $g^{\boldsymbol{A}}_{\vec{T},U}$ on $A_{\vec{S}}$ [22, Page 10].

This kind of model is related to the *ideal model* used by MacQueen *et al.* [38, 39, 40]. It was originally designed to deal with recursive types, but was adapted by Cardelli and others to give a semantics to models of object-oriented languages [7].

4.4 Category-Sorted Algebras

If the carrier sets of all types are disjoint from the carrier sets of every other type, including their supertypes, then one can assign unique types to values in the algebra's carrier set. The notion that somehow, values of subtype objects are similar to values of supertype objects can be captured by using coercion functions. These coercion functions map values of a subtype into the carrier set of their supertype. This is the idea behind Reynolds's "category-sorted algebras" [45, 46].[3]

Definition 4.5 *Let* $\Sigma = (TYPE, \leq, OP, ResType)$ *be a signature with subtyping. A pair* $(\boldsymbol{A}, c)$, *is a* category-sorted Σ-algebra *if and only if*

- $\boldsymbol{A}$ *is a subtype polymorphic* Σ*-algebra with disjoint carrier sets for each type,*
- *for all* $g \in OP$, $\vec{T} \in TYPE^*$, *and* $U \in TYPE$, *if* $ResType(g, \vec{T}) = U$, *then* $g^{\boldsymbol{A}} : A_{\vec{T}} \to A_U$, *and*
- $c = \langle c_{S \to T} : S \in TYPE, T \in TYPE, S \leq T \rangle$ *is a family of coercion functions such that*
  - *whenever* $S \leq T$, $c_{S \to T} : A_S \to A_T$,
  - $c_{T \to T}$ *is the identity on* A_T,

[3] Reynolds does not require explicitly that the carrier sets be disjoint, although the machinery seems better motivated with this condition. In addition, Reynolds's definition is stated in terms of category theory and is thus more concisely stated than ours and in some ways more general. Our definition can be seen as a restriction of his to simple categories, in which there is only one way in which one type may be a subtype of another, and consequently a unique coercion.

- *whenever* $S \leq T$ *and* $T \leq U$, *then* $c_{S \to U} = c_{T \to U} \circ c_{S \to T}$,
- *and the following functorial property is satisfied. Whenever* $U \in TYPE$, $ResType(g, \vec{T}) = U$, $\vec{S} \leq \vec{T}$, $ResType(g, \vec{S}) = U'$, *and* $\vec{o} \in A_{\vec{S}}$, *then*

$$c_{U' \to U}(g(\vec{o})) = g(c_{\vec{S} \to \vec{T}}(\vec{o})).$$

The functorial property can be thought of in several ways. One way is a specification of the conditions that an operation must satisfy for a subtype. If the coercions are invertible, then one can also use the functorial property to actually define what the operations of a subtype do. That is, whenever $U \in TYPE$, $ResType(g, \vec{T})$ is defined, $\vec{S} \leq \vec{T}$, and $\vec{o} \in A_{\vec{T}}$, then

$$g(\vec{o}) = c^{-1}_{U' \to U}(g(c_{\vec{S} \to \vec{T}}(\vec{o}))) \tag{34}$$

When the coercions are invertible, this equation may thus be taken as a simple model of method inheritance.

4.5 Comparisons

A starting place for comparing order-sorted and category-sorted algebras is the fact that a subset relationship can be modeled by a coercion function. That is, if $A_S \subseteq A_T$, then there is an identity injection $i_{S \to T}$ defined by $i_{S \to T}(o) = o$ that can act as a coercion function [45, Page 217]. Using these as the coercion functions almost makes an order-sorted algebra into a category-sorted algebra, but our definition of a category-sorted algebra also requires that the carrier sets of each type be disjoint.

To accomplish this, for each signature $\Sigma = (TYPE, \leq, OP, ResType)$ we define the function $osa2csa_\Sigma$ as follows. This function makes the carrier sets of the various types disjoint by tagging them with their type. The coercion functions simply adjust the type tag, and the operation interpretations ignore the type tag.

$osa2csa_\Sigma(\langle A_T : TYPE \rangle, OP^{\boldsymbol{A}}) = ((A', OP^{\boldsymbol{A}'}), i')$
where $A' = \langle A'_T : TYPE \rangle$
$A'_T = \{(T, v) : v \in A_T\}$
$OP^{\boldsymbol{A}'} = \{g^{\boldsymbol{A}'} : g \in OP\}$
$g^{\boldsymbol{A}'}((T_1, o_1), \ldots, (T_n, o_n)) = (ResType(g, (T_1, \ldots, T_n)), g^{\boldsymbol{A}}(o_1, \ldots, o_n))$
$i' = \langle i'_{S \to T} : S \in TYPE, T \in TYPE, S \leq T \rangle$
$i'_{S \to T}(S, o) = (T, o)$

This definition gives the following.

Lemma 4.6 *Let* Σ *be a signature with subtyping. Let* $\boldsymbol{A}$ *be an order-sorted* Σ*-algebra. Then* $osa2csa_\Sigma(\boldsymbol{A})$ *is a category-sorted* Σ*-algebra.* ∎

Thus from an order-sorted algebra, one can immediately obtain a category-sorted algebra.

The converse construction is very useful for our model theoretic studies of behavioral subtyping. It can be accomplished in two steps.

First, a category-sorted algebra can be made into a subtype polymorphic algebra by simply forgetting the coercion functions. This fulfills the definition of a subtype polymorphic algebra trivially.

Second, and more interestingly, we can make an arbitrary subtype polymorphic algebra into an order-sorted algebra. This is accomplished by defining the carrier set of each type of the order-sorted algebra to be the union of the carrier set of that type and all its subtypes in the original algebra.

For a given signature $\Sigma = (TYPE, \leq, OP, ResType)$, we define the following function to do this translation.

$$toOSA_{\Sigma}(A, \{g^{\boldsymbol{A}} : g \in OP\}) = (\langle \hat{A}_T : T \in TYPE \rangle, \{g^{\boldsymbol{A}} : g \in OP\})$$

Then by definition of order-sorted algebra we have the following.

Corollary 4.7 *Let $\Sigma = (TYPE, \leq, OP, ResType)$ be a signature. If $\boldsymbol{A}$ be a subtype polymorphic Σ-algebra, then $toOSA(\boldsymbol{A})$ is an order-sorted Σ-algebra.* ∎

To summarize, it is possible to translate each kind of algebraic model into the others. The translation to order-sorted algebras is a semantic counterpart to the subsumption rule of various type systems (e.g., [7]), in which if o has type S and $S \leq T$, then o has type T. The of translation to category-sorted algebras is a semantic counterpart to the implementation of OO languages, in which each object typically has a unique type tag. In terms of proving properties of OO systems, subtype polymorphic algebras impose the least restrictive conditions on the construction of algebras, and so may be preferable for that reason.

5 Conclusions

In this paper we have tried to relate several models of objects. We first related static, class-based record models to class-based generic function models. The generic function models can be seen as a rearrangement of the information in the record models. However, this rearrangement has some practical advantages for programming [4, 11].

While the generic function models have various advantages, it is important to note the assumptions behind our simulation of the record model by the generic function model. We believe that the simulation is only possible for a static, class-based record model. Although we have no proof of this, there seems to be no way to find all the methods without having a statically-known set of classes. There also seems to be no way for a generic function to distinguish between objects that should behave differently but which have the same data, unless objects are tagged with their some information (their class) which tells what method should be used. Note that the class of an object, since it must be statically known, cannot be the object's identity, otherwise the assumption about classes being statically-known would be violated.

We also related the type systems of the record and generic function models. Our simulation of the record-based models was shown to preserve typing. This is another indication of how straightforward it is for the generic function model to simulate the static, class-based record model. It also points out the similarities of the two type systems. In particular, the rule for subtyping generic function types is closely related to the rule for subtyping method dictionaries.

An interesting difference between type systems for the record model and the generic function model is that the generic function model seems better suited to by-name type checking and subtyping. For the record model, one can use either structural or by-name type checking and subtyping. However, in the generic function model, it seems difficult to decide subtyping structurally, because there is no easy way to obtain useful information about the methods that apply to an object, which is what is used to do structural subtyping in the record model. This reflects the fact that objects in the generic function model are not self-interpreting.

To summarize, it appears that single-dispatching and multiple dispatching languages, while closely related, each have advantages that are not offered by the others. This suggests to the language designer that perhaps some hybrid might be advantageous. The ability of the generic function model to dispatch on tuples, and the record model's lack of dispatch on tuples hints at one way the two mechanisms might be grafted together.

Algebraic models related to Goguen and Meseguer's order-sorted algebras and Reynolds's category-sorted algebras are best seen as abstractions of the generic function models. However, because the generic function model can easily simulate static, class-based record models, algebraic models can also be seen as abstractions of standard single-dispatching languages.

The category-sorted and order-sorted algebras turn out to be easily translatable into each other. Category-sorted algebras retain the flavor of class-based OO languages in that if objects are tagged with their class, then the carrier sets of each class are disjoint. Order-sorted algebras embody the idea that, since the objects of a type's subtypes can all act like objects of that type, effectively a supertype's carrier set contains its subtype's carrier sets. Subtype polymorphic algebras can be seen as a common abstraction of these two kinds of models.

Acknowledgements

Thanks to Todd Millstein for comments on an earlier draft, and for suggesting that we make our comparisons between the record and generic function models constructive, which greatly improved the paper. Thanks to Todd and Craig Chambers for many discussions about multimethod languages and their type systems.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, N.Y., 1996.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[4] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[5] Kim B. Bruce. The equivalence of two semantic definitions for inheritance in object-oriented languages. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Mathematical Foundations of Programming Semantics, 7th International Conference, PA, USA, March 1991, Proceedings*, volume 598 of *Lecture Notes in Computer Science*, pages 102–124. Springer-Verlag, New York, N.Y., 1992.

[6] Kim B. Bruce and Peter Wegner. An algebraic model of subtype and inheritance. In Francois Bançilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, pages 75–96. Addison-Wesley, Reading, Mass., August 1990.

[7] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[9] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston, 1997.

[10] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995. A preliminary version appeared in *ACM Conference on LISP and Functional Programming*, June 1992 (pp. 182–192).

[11] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, N.Y., 1992.

[12] Craig Chambers. The Cecil language specification and rationale: Version 2.0. Available from http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html, December 1995.

[13] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. *TOPLAS*, 17(6):805–843, November 1995.

[14] Craig Chambers and Gary T. Leavens. BeCecil, a core object-oriented language with block structure and multimethods: Semantics and typing. Technical Report 96-17a, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, April 1997. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu. Also University of Washington Department of Computer Science and Engineering TR number UW-CSE-96-12-02.

[15] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *ACM SIGPLAN Notices*, 24(10):433–443, October 1989. OOPSLA '89 Conference Proceedings, Norman Meyerowitz (editor), October 1989, New Orleans, Louisiana.

[16] William R. Cook. A denotational semantics of inheritance. Technical Report CS-89-33, Department of Computer Science, Brown University, Providence, Rhode Island, May 1989.

[17] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[18] Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl/locate/entcs/volume1.html.

[19] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a taxonomy to constructive proposals and their validation. *ACM SIGPLAN Notices*, 27(10):201–217, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[20] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, N.Y., 1985.

[21] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[22] Joseph A. Goguen and Jose Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions, and partial operations. Technical Report SRI-CSL-89-10, Computer Science Laboratory, SRI International, July 1980.

[23] Joseph A. Goguen and Jose Meseguer. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. In *Symposium on Logic in Computer Science, Ithaca, NY*, pages 18–29. IEEE, June 1987.

[24] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation*. Addison-Wesley Publishing Co., Reading, Mass., 1983.

[25] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[26] J. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[27] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. *ACM SIGPLAN Notices*, 21(11):347–349, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[28] Samuel Kamin. Inheritance in smalltalk-80: A denotational definition. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 80–87. ACM, January 1988.

[29] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[30] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[31] Gary T. Leavens and Don Pigozzi. Typed homomorphic relations extended with subtypes. In Stephen Brookes, editor, *Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 144–167. Springer-Verlag, New York, N.Y., 1992.

[32] Gary T. Leavens and Don Pigozzi. An exact algebraic characterization of behavioral subtyping. Technical Report 96-15, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 1996. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[33] Gary T. Leavens and Don Pigozzi. The behavior-realization adjunction and generalized homomorphic relations. *Theoretical Computer Science*, 177:183–216, 1997.

[34] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, October 1990.

[35] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[36] Gary Todd Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989. The author's Ph.D. thesis.

[37] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–223, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[38] D. B. MacQueen and Ravi Sethi. A semantic model of types for applicative languages. In *ACM Symp. on LISP and Functional Programming*, pages 243–252. ACM, 1982.

[39] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah*, pages 165–174. ACM, January 1984.

[40] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, Oct./Nov. 1986.

[41] Narciso Marti-Oliet and Jose Meseguer. Inclusions and subtypes. Technical Report SRI-CSL-90-16, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, Calif., December 1990.

[42] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[43] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 109–124. ACM, January 1990.

[44] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 1993.

[45] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, January 1980.

[46] John C. Reynolds. Three approaches to type structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, New York, N.Y., March 1985.

[47] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass., 1986.

[48] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

[49] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for self. In Walter Olthoff, editor, *ECOOP '95 - Object-Oriented Programming 9th European Conference, Aarhus, Denmark*, number 952 in Lecture Notes in Computer Science, pages 303–330. Springer-Verlag, New York, N.Y., 1995.

[50] Bjarne Stroustrup. *The C++ Programming Language: Third Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1997.

[51] David Ungar and Randall B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–241, December 1987. OOPSLA '87 Conference Proceedings, Norman Meyrowitz (editor), October 1987, Orlando, Florida.

[52] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76. ACM, January 1989.